Une architecture minimisant les échanges entre processeur et mémoire

Florent de Dinechin, Maxime Darrin, Antonin Dudermel, Sébastien Michelland, Alban Reynaud

Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Une architecture minimisant les échanges entre processeur et mémoire

Florent de Dinechin, Maxime Darrin, Antonin Dudermel, Sébastien Michelland, Alban Reynaud

Résumé
Dans une architecture de von Neumann, les échanges de données représentent le gros de l'énergie dépensée. Or le processeur communique avec la mémoire au moyen d'un bus d'adresse et d'un bus de données de grandes tailles (entre 8 et 64 bits). Cette granularité contraint ces échanges, et en particulier l'encodage des instructions du processeur. Cet article étudie ce qui est possible en levant cette contrainte. Il propose une architecture 64 bits dont la mémoire est adressable par bit, ce qui permet des instructions de taille arbitraire. Pour ne pas devoir envoyer une adresse complète à la mémoire à chaque accès, la solution proposée est l'usage de pointeurs auto-incrémentés dupliqués dans la mémoire et le processeur.
Cet article décrit aussi une expérience pédagogique réalisée à l'ENS-Lyon. Un premier jeu d'instruction a été défini en TD et son encodage choisi à la main. Ceci a permis aux étudiants d'écrire en binôme un assembleur et un simulateur, puis plusieurs milliers de lignes de programmes allant du petit noyau de calcul au jeu vidéo et à l'émulateur. Sur les traces de ces programmes, on a pu ensuite calculer un encodage optimal des instructions en fonction de leur fréquence, et les comparer à l'encodage initial. Cette étude a aussi porté sur la quantité de bits transférés entre processeur et mémoire.

1. Introduction et motivation

Cet article s'intéresse à l'encodage du jeu d'instruction d'un processeur de von Neumann, interfacé à une mémoire par un bus d'adresse et un bus de données (figure 1).
Le jeu d'instruction d'un tel processeur (Instruction Set Architecture ou ISA) reflète l'état de la loi de Moore à l'époque de sa conception. La mémoire intégrée sur une puce double tous les deux ans, et le bus mémoire suit : sa taille (w_a sur la figure) grandit d'un bit tous les deux ans. Ceci se traduit à son tour par une croissance de la taille des registres qui, dans un processeur, peuvent adresser la mémoire. Toutefois, cette croissance a suivi des puissances de 2 : de 8 bits

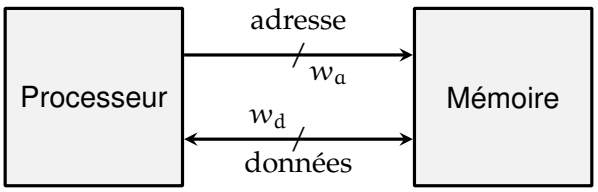

FIGURE 1 – Une machine de von Neumann.

aux temps héroïques à 16 bits dans les années 1970, 32 bits dans les années 1980, et 64 bits dans les années 2000, et sans doute pour encore quelques dizaines d'années.
Les ISAs définissent aussi la granularité à laquelle on peut adresser la mémoire. L'ancêtre 4004 et le processeur Saturn des calculettes HP adressaient leur mémoire par case de 4 bits : ceci correspondait au besoin du décimal codé en binaire. La plupart des processeurs (dont les Intel/AMD de nos PC et les ARM de nos gadgets) adressent leur mémoire par octet pour être efficaces sur le traitement de texte.
Ceci nous amène à la première constatation. Dès lors que les adresses font 64 bits, on peut parfaitement décider que le adresses seront des adresses de bits. On perd un facteur 8 en quantité de mémoire adressable, ce qui aurait été inacceptable à l'époque des adresses sur 16 bits, mais est actuellement indolore : avec un espace d'adressage sur 64 bits, il reste tout de même $2^{61} \approx 2.10^{18}$ bits, ou 2 exabits, à adresser.
On parle ici de l'abstraction offerte par le jeu d'instruction, pas de l'interface physique. Les deux sont déjà différentes dans les processeurs actuels. Si l'ISA définit une mémoire adressable par octet, l'accès physique à la mémoire se fait à travers un cache qui, côté processeur, offre l'adressage par octet, mais côté mémoire physique réalise des accès alignés sur une grande puissance de 2 correspondant à la taille de la ligne de cache. On peut estimer que passer d'un adressage par octet à un adressage au bit ajoute trois niveaux de multiplexeurs au décodage de la ligne de cache, un changement plus quantitatif que qualitatif.
Ce qu'on gagne à avoir une adresse pour chaque bit, c'est que la taille des données et des instructions peut être variable au bit près : on peut espérer n'envoyer que les bits utiles. Comme exemple d'application, on peut citer les UNUM, un format de représentation des nombres flottants de taille variable et auto-descriptif (les tailles d'exposant et de mantisse sont encodées dans un en-tête) proposé par Gustafson en 2015 [6]. Cette proposition est déjà abandonnée par son auteur [8] malgré quelques bonnes idées, mais une de ses motivations reste : *fetching operands costs more than computing on them* [4]. Il y a un facteur 1000 entre l'énergie qu'il faut pour calculer une opération flottante 64bits et l'énergie qu'il faut pour aller chercher ses opérandes en RAM. Cela justifie qu'on cherche à minimiser les échanges avec la RAM.
Il y a aussi des applications, comme les réseaux de neurones, qui se contentent très bien de données sur de très petites précision, voire des données binaires [3, 1, 2, 7].
Attention, il ne faut pas que l'on ait à envoyer 64 bits d'adresse pour chaque bit de données échangé, ce qui serait bien pire que le faire pour chaque octet. Reprendre la figure 1 en fixant juste $w_d = 1$ serait contre-productif.
Ce travail essaye d'imaginer une architecture (au sens ISA) dans laquelle on peut tailler instructions et données au bit près, et ne payer que pour les bits effectivement transférés. Il a été réalisé dans le cadre de l'enseignement de l'architecture en L3 à l'ENS-Lyon, où chaque année les étudiants doivent définir et implémenter une ISA originale sous des contraintes un peu arbitraires. Certains détails techniques manquant à cet articles pour faute de place pourront être trouvés dans les supports des travaux dirigés correspondant :
`perso.citi-lab.fr/fdedinec/enseignement/2017/ASR1/`.

2. Architecture générale et interface mémoire

Pour concrétiser l'adressabilité au bit près, nous proposons une interface processeur-mémoire strictement série représentée sur la figure 2. Les données passent en série, bit à bit, sur le fil `D`. Le processeur contrôle le sens de transfert par les fils `Read` et `Write`. Pour ne pas avoir à transmettre une adresse de n bits par bit de donnée, les adresses sont le plus souvent implicites. Dans ce but, on a 4 compteurs, répliqués dans le processeur et la mémoire. Le pointeur de

programme, PC, est l'un de ces compteurs. Il s'auto-incrémente, classiquement, au cours de la lecture d'une instruction, et pour passer à l'instruction suivante.
Pour lire une donnée, le processeur sélectionne un des 4 compteurs en positionnant les deux bits `Select`. Puis il lève `Read`. Tant que `Read` vaut 1, le compteur sélectionné s'incrémente (côté processeur comme côté mémoire) et les bits qu'il pointe sont transférés sur le fil `D`. Lorsque le processeur a lu toute une donnée, il baisse `Read`. Le mécanisme est similaire en cas d'écriture.
Les instructions de lecture/écriture mémoire ne donnent pas d'adresse, mais précisent un compteur (encodé sur 2 bits) et le nombre de bits mémoire à lire/écrire (parmi 1, 4, 8, 16, 32 et 64 bits, encodé par un code prefix-free sur 2 ou 3 bits). Il faut aussi des instructions qui permettent de transférer un registre dans un compteur ou inversement. Ces instructions ont un encodage compact (le code op, suivi du registre encodé sur 3 bits et du compteur encodé sur 2). Par contre, leur exécution provoque bien plus de transferts de bits : par exemple pour changer un compteur, le processeur lève `RWCounter`, puis transfère sur `D` les bits du compteur sélectionné par `Select`, poids faibles en tête. Lorsque le processeur baisse `RWCounter`, la mémoire complète le compteur avec le dernier bit transmis (extension de signe) [1]. Si le compteur était le PC, on a réalisé un saut.
Cette interface est identique pour un processeur 32 ou 64 bits, et ne dépend pas non plus de la taille mémoire.
Nous sommes conscients que cette interface n'est pas réaliste, en tout cas pas avec une horloge à haute fréquence dans une technologie moderne : Il y a une hypothèse de synchronicité des signaux de contrôle et de donnée qu'on ne se permet plus dans les liens série rapide ou les bus d'interface mémoire. Nous l'utilisons essentiellement comme une règle du jeu pour étudier quantitativement la question de la taille de l'encodage d'un jeu d'instructions.
Pour le reste, le processeur embarque un cœur RISC très classique, d'architecture load/store, à 8 registres de 64 bits. Le choix de 8 registres seulement est arbitraire et discutable : il est motivé par des raisons pédagogiques (faire en sorte que les étudiants soient à l'étroit dans les registres dès l'écriture de programmes simples) et par une raison pratique : encoder le numéro de registre sur aussi peu de bits que possible. Passer le nombre de registres de 8 à 16 ajouterait 1, 2 ou 3 bits à l'encodage de la plupart des opérations.

1. Ce fonctionnement sera raffiné, pour des raisons de performance, en section 4

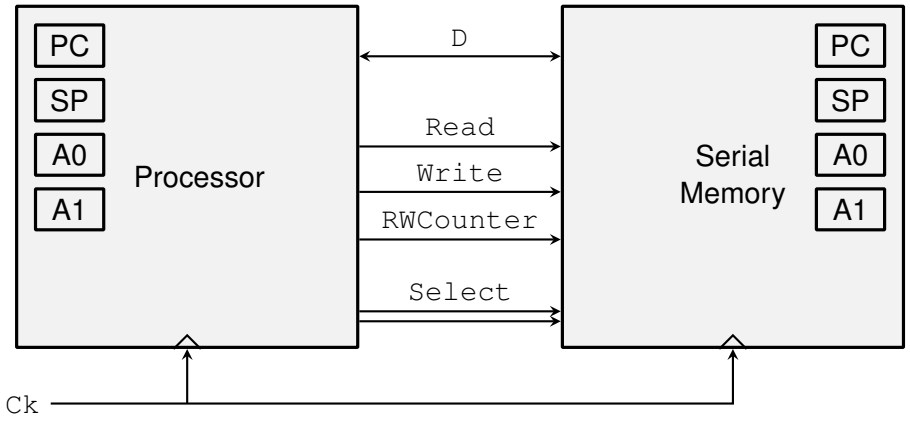

FIGURE 2 – L'interface processeur-mémoire proposée

3. Présentation du jeu d'instruction

Le jeu d'instruction décrit dans cette section est l'aboutissement des deux séances de TD dans lesquelles on a essayé de discuter les tenants et les aboutissants de chaque décision. Les choix faits ne sont pas toujours ceux qui étaient considérés comme les meilleurs : nous avons aussi privilégié la faisabilité dans le temps limité consacré au module d'ASR1.
La figure 3 montre un exemple de programme et son encodage. Les instructions commencent toutes par un code opération (opcode), suivi des éventuels opérandes. Les mnémoniques sont écrits exactement dans le même ordre (figure 3). Une fois que le processeur a reçu l'opcode, il sait combien d'opérande il doit recevoir et quel est leur type.
La liste complète des instructions est donnée dans la table 3 en annexe. Les instructions ALU viennent en version 2 et 3 opérandes, la destination venant toujours en premier. Par exemple,

`add2 r0 r1` réalise $r_0 \leftarrow r_0 + r_1$.
`add3 r0 r1 r2` réalise $r_0 \leftarrow r_1 + r_2$.

Dans les programmes écrits à la main, il faut constater qu'on a utilisé surtout les versions à 2 opérandes. Ceci est similaire à l'évolution du jeu d'instruction ARM vers le mode Thumb2 qui permet d'encoder sur 16 bits des instructions à deux opérandes, et sur 32 bits des instructions plus puissantes à 3 ou 4 opérandes.
L'opcode est différent pour les opérations dont les deux opérandes sont des registres (par exemple `add2`), et les opérations portant sur un registre et une constante (par exemple `add2i`). Il y a des opcodes de différentes tailles (un opérande n'est jamais le préfixe d'un autre, ainsi le décodage n'est jamais ambigu).
Les opérandes peuvent également être de différentes tailles. En particulier, les constantes commencent par un préfixe qui décrit le nombre de bits sur lequel est encodée la constante (table 1). Il s'agit d'un genre ad-hoc de Gamma-coding [5]. Ainsi les petites constantes courantes comme 0 et 1 sont encodées sur peu de bits, alors que les grandes constantes son également encodables. Pour les sauts relatifs, on paiera moins cher les sauts proches (jusqu'à +/- 128 bits, soit une dizaine d'instructions) que les sauts plus distants, qui sont par définition pris moins fréquemment. On a un encodage des constantes différent pour chaque classe d'instruction (table 1), puisque les besoins ne sont pas les même.
Les instructions de branchement sont relativement classiques : saut, saut conditionnel sur drapeau, saut à une sous-routine (l'adresse de retour étant stockée dans le registre `r7`).

étiquette	mnémonique	encodage initial	encodage Huffman de l'opcode
	`leti r0 17`	`0111 000 1000010001`	`100 000 1000010001`
	`leti r1 42`	`0111 001 1000101010`	`100 001 1000101010`
	`leti r2 0`	`0111 010 00`	`100 010 00`
`nonzero:`	`shift right r0 1`	`1000 1 000 1`	`00 1 000 1`
	`jumpif nc next`	`1011 101 000001010`	`01 101 000001010`
	`add2 r2 r1`	`0000 010 001`	`1010 010 001`
`next:`	`shift left r1 1`	`1000 0 001 1`	`00 0 001 1`
	`cmpi r0 0`	`0101 000 00`	`11 000 00`
	`jumpif nz nonzero`	`1011 001 010111011`	`01 001 011000101`
`loop:`	`jump loop`	`1010 011110011`	`10110 011110010`

FIGURE 3 – Un exemple de programme assembleur (une multiplication entière)

TABLE 1 – Encodage *prefix-free* des différentes constantes

addr adresses, déplacements	*const* constantes ALU	*shiftval* constantes de shift	*size* tailles	
0 + 8 bits	0 + 1 bit	0 + 6 bits	00 : 1 bit	01 : 4 bits
10 + 16 bits	10 + 8 bits	1 (constante 1)	100 : 8 bits	101 : 16 bits
110 + 32 bits	110 + 32 bits		110 : 32 bits	
111 + 64 bits	111 + 64 bits		111 : 64 bits	

4. Transfert des bits à l'exécution

Au dela de l'encodage des instructions, nous avons aussi cherché à minimiser les transferts de bits à l'exécution. Par exemple, considérons la manipulation d'une pile descendante en mémoire. Si le `pop` marche tout seul avec un pointeur de pile autoincrémenté, le `push`, lui, doit écrire aux adresses *précédant* `SP`. Ceci peut se faire par l'équivalent d'une instruction `setctr` pour descendre `SP`, d'une écriture autoincrémentée, et d'un second `setctr` qui redescend `SP`. Un `push` de 64 bits doit ainsi transférer deux adresses en plus des 64 bits. Nous avons résolu ceci en ajoutant une écriture prédécrémentée, déclenchée lorsque `Read` et `Write` sont tous deux à 1.
Un autre problème qui est apparu est que le plus souvent, on veut mettre à jour un compteur avec une valeur proche de celle qu'il a déjà, par exemple le `PC` lors d'un saut relatif, ou un incrément de pointeur dans un tableau. Il est naturel de vouloir envoyer seulement la différence entre l'ancienne et la nouvelle valeur. Lorsque `RWCounter` vaut 1, ce qui passe sur `D` est donc toujours une différence, poids faible en premier. Lorsque `RWCounter` est baissé, la valeur envoyée subit une extension de signe sur 64 bits, et cette valeur est ajoutée au compteur. Dans les tests que nous présentons ci-dessous, cette technique a permis de gagner un facteur dépassant 10 sur le nombre total de bits transmis pour mettre a jour les compteurs.

5. Expérimentations

Nous n'avons pas encore de compilateur pour cette architecture. Les expériences, dont quelques résultats sont rapportés dans la table 2, portent sur une petite suite de benchmarks écrits en assembleur à la main :
Multiplication entière Le choix d'une ISA sans instruction de multiplication ni division est essentiellement pédagogique, pour faire manipuler l'arithmétique de bas niveau. les étudiants ont donc du écrire le programme de la figure 3, et même l'assembler à la main dans un premier temps. Il n'y a pas d'accès mémoire dans ce benchmark, hors du programme lui-même. Pour comparaison, la troisième ligne de la table décrit la taille et le nombre de bits échangés à l'exécution par un programme très similaire dans l'assembleur du le processeur RISC 16 bits MSP430 qui a un jeu d'instruction comparable : notre proposition est plus compacte et transmet moins de bits à l'exécution (essentiellement par l'économie du transfert des adresses).
Produit de matrices C'est un produit de matrices 32x32 d'entiers 64 bits aléatoires. Ce programme utilise la multiplication entière précédente, mais les matrices sont lues et écrites en mémoire. On présente deux simulations : la première avec des matrices pleines, et la seconde avec des matrice creuse (en moyenne 90% de 0). Les multiplications par 0 terminant instantanément, mais chaque 0 étant tout de même stocké en mémoire comme 64 bits, cette seconde version provoque un rapport bits de données / bits de code plus élevé.

	taille du programme			bits échangés à l'exécution					
benchmark	instr	bits	BPI	prog	data R	data W	counters	branch	total
binmult I	10	125	12.5	89.4%				10,6%	415
H		113	11.3	85.5 %				14.5 %	373
msp430	*10*	*192*	*19.2*	–	–	–	–	–	*960*
matmul I	112	1632	14.6	80.4 %	5.0 %	2.6 %	0.3 %	11.8 %	1.72e8
(dense) H	112	1817	16.2	75.4 %	5.6 %	2.9 %	0.3 %	15.9 %	1.54e8
matmul I	112	1632	14.6	55.3 %	23.3 %	12.1 %	1.2 %	8.0 %	3.68e7
(sparse) H	112	1613	14.4	52.1 %	25.2 %	13.1 %	1.3 %	8.3 %	3.41e7
Chip8 I	768	14155	18.4	64.3 %	10.4 %	9.7 %	7.5 %	8.1 %	1.068e8
H	768	13699	17.8	63.1 %	10.7 %	9.9 %	7.7 %	8.6 %	1.063e8

TABLE 2 – Nombre de bits échangés par benchmark. BPI : bits par instruction (moyen). Encodage des opcodes : I initial, H Huffman. Les colonne à partir de *data R* comptent les bits qui passent sur D à l'exécution des instructions de lecture, d'écriture, d'accès aux compteurs, et de sauts/call respectivement.

Chip8 Un binôme a écrit un émulateur CHIP8 [9] pour notre système. Ce benchmark fait tourner le jeu BRIX pendant exactement une minute par partie. Ce benchmark est très orienté mémoire puisqu'il interagit beaucoup avec l'écran.

La première expérience consiste à utiliser l'encodage initial. Des compteurs, dans le simulateur, mesurent l'utilisation de chaque instruction.D'autres compteurs comptent les bits échangés sur les fils à l'exécution. Les résultats correspondants sont donnés dans la première ligne pour chaque benchmark dans la table 2.
La seconde expérience (dont les résultats sont donnés sur la seconde ligne pour chaque benchmark dans la table 2) consiste à changer l'encodage initial des opcodes pour un encodage de Hufman calculé sur la trace d'exécution de la première expérience, donc optimal *pour ce benchmark*. L'intérêt de cette seconde expérience est de donner une borne inférieure : nos trois benchmarks réclament un encodage optimal assez différents, et l'idée de changer l'encodage des instructions en fonction du programme va au-delà de la présente étude. Du reste, si l'on voit que l'encodage des opcodes peut encore être amélioré, les gains à espérer restent faibles.

6. Conclusion et perspectives d'améliorations

Les premiers résultats de cette étude sont encourageants, puisqu'on arrive à des encodages d'instructions en moins de 16 bits de moyenne, opérandes compris. Nous proposons donc une ISA 64 bits dont la taille de code est compétitive avec celle de la génération 8 bits ou des microcontroleurs récents 16 bits. Nous disposons d'un assembleur et d'un simulateur dans lesquels l'encodage des opcodes est paramétrable, ce qui permet aussi d'ajouter facilement des instructions.
Cette approche s'accomode fort bien d'instructions très disparates en termes d'opérandes. Ainsi, ajouter une instruction de multiplication-accumulation à 4 arguments ne nous posera pas de problème particulier – étendre notre ISA avec de telles instructions fait partie des pistes à explorer prochainement.
Ceci fera grossir nos opcodes : pour un jeu d'instruction vraiment complet (incluant virgule flottante, interruptions, etc), nous arriverons peut-être à la conclusion que le *bytecode*, l'adressa-

bilité à l'octet, est décidément le bon compromis. C'est celui de l'ISA dominante, où l'instruction `ret` s'encode toujours en 8 bits, et aussi des machines virtuelles comme Java ou Python, même si (à notre connaissance) ce choix découle plus de raisons historiques que de savantes études.
Nous nous somme concentrés sur l'encodage des instructions, et avec raison : dans nos expériences, le code représente toujours l'essentiel des bits échangés.
Nous n'avons vraiment fait varier que l'opcode : l'encodage des registres reste naïf, l'encodage des constantes décrit par la table 1 est toujours celui décidé en TD à l'intuition. Pour mener une étude plus poussée, il faut une base de *benchmarks* plus étendue, donc un compilateur. Ce sera l'objet d'un projet l'année prochaine.
Pour cela, il faudra des instructions d'accès indexé à la pile (SP+constante) pour permettre d'accéder efficacement à l'enregistrements d'activation d'une procédure. Dans ce cas, on a envie de déléguer à la mémoire l'addition de SP et de la constante, puisqu'elle est présente en mémoire dans le code : pourquoi la faire transiter sur D dans les deux sens ? Il en va du reste de même pour un saut relatif (addition sur PC d'une constante qui est présente en mémoire dans le code).
La question se pose aussi de la pertinence de ce travail pour construire un "vrai" processeur. Nous sommes conscients que notre lien série maître/esclave entre processeur et mémoire n'est plus pertinent depuis plusieurs dizaines d'années. De ce point de vue, ce travail reste théorique. Lui trouvera-t-on une niche applicative qui justifierait de se lancer dans la construction de ce processeur ? Serait possible d'abriter notre ISA derrière un cache classique, en espérant que la compacité du jeu d'instruction permet d'économiser sur celui-ci ? Il faudrait pour cela montrer que cette économie n'est pas anéantie par le surcoût du décodage. Peut-être que la réponse à cette question passe par une implémentation du processeur.

Bibliographie

1. Alemdar (H.), Leroy (V.), Prost-Boucle (A.) et Petrot (F.). – Ternary neural networks for resource-efficient AI applications. – In *International Joint Conference on Neural Networks (IJCNN)*, pp. 2547–2554, May 2017.
2. Amiri (M.), Hosseinabady (M.), McIntosh-Smith (S.) et Nunez-Yanez (J.). – Multi-precision convolutional neural networks on heterogeneous hardware. – In *Design Automation and Test in Europe*, 2018.
3. Andri (R.), Cavigelli (L.), Rossi (D.) et Benini (L.). – YodaNN : An ultra-low power convolutional neural network accelerator based on binary weights. – In *Annual Symposium on VLSI*, pp. 236–241. IEEE, 2016.
4. Dally (B.). – GPU computing : To exascale and beyond. – In *SuperComputing*, 2010. – `www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf`.
5. Elias (P.). – Universal codeword sets and representations of the integers. *Transactions on Information Theory*, vol. 21, n2, mars 1975, pp. 194–203.
6. Gustafson (J. L.). – *The End of Error : Unum Computing*. – Chapman Hall, *CRC Computational Science*.
7. Preußer (T. B.), Gambardella (G.), Fraser (N.) et Blott (M.). – Inference of quantized neural networks on heterogeneous all-programmable devices. – In *Design Automation and Test in Europe (DATE)*, 2018.
8. Tichy (W.). – Unums 2.0 : An Interview with John L. Gustafson. *Ubiquity*, vol. 2016, n09, septembre 2016, pp. 1 :1–1 :16.
9. Weisbecker (J.). – An easy programming system. *BYTE magazine*, décembre 1978, pp. 108–122. – Voir aussi `https://en.wikipedia.org/wiki/CHIP-8`.

TABLE 3 – Annexe : Liste des instructions avec leur encodage initial.

Le choix des opcodes est arbitraire. Les opérandes d'une instruction suivent l'opcode :

- *reg* ∈ {`r0`, `r1`, ..., `r7`} et est encodé par le numéro du registre en binaire.
- *const*, *shiftval* et *addr* sont définis par la table 1. La colonne *ext* de la table 3 précise si une constante est étendue avec son signe (s) ou des zéros (z).
- *cond* est une condition portant sur les drapeaux et encodée sur 3 bits.
- *ctr* est un des 4 compteurs de la figure 2, encodé sur 2 bits.
- *dir* peut être `left`, encodé par 0, ou `right`, encodé par 1.

opcode	mnemonic	operands	description	ext.	MàJ flags
0000	`add2`	*reg reg*	addition		zcvn
0001	`add2i`	*reg const*	add immediate constant	z	zcvn
0010	`sub2`	*reg reg*	subtraction		zcvn
0011	`sub2i`	*reg const*	subtract immediate constant	z	zcvn
0100	`cmp`	*reg reg*	comparison		zcvn
0101	`cmpi`	*reg const*	comparison with immediate constant	s	zcvn
0110	`let`	*reg reg*	register copy		
0111	`leti`	*reg const*	fill register with constant	s	
1000	`shift`	*dir reg shiftval*	logical shift		zcn
10010	`readze`	*ctr size reg*	read *size* mem. bits (zero-extended) to *reg*		
10011	`readse`	*ctr size reg*	read *size* mem. bits (sign-extended) to *reg*		
1010	`jump`	*addr*	relative jump		
1011	`jumpif`	*cond addr*	conditional relative jump		
110000	`or2`	*reg reg*	logical bitwise or		zcn
110001	`or2i`	*reg const*	logical bitwise or	z	zcn
110010	`and2`	*reg reg*	logical bitwise and		zcn
110011	`and2i`	*reg const*	logical bitwise and	z	zcn
110100	`write`	*ctr size reg*	write the lower *size* bits of *reg* to mem		
110101	`call`	*addr*	sub-routine call	s	
110110	`setctr`	*ctr reg*	store *reg* to a counter		
110111	`getctr`	*ctr reg*	copy the current value of counter to *reg*		
1110000	`push`	*size reg*	push value of register on stack		
1110001	`return`		return from subroutine		
1110010	`add3`	*reg reg reg*			zcvn
1110011	`add3i`	*reg reg const*		z	zcvn
1110100	`sub3`	*reg reg reg*			zcvn
1110101	`sub3i`	*reg reg const*		z	zcvn
1110110	`and3`	*reg reg reg*			zcn
1110111	`and3i`	*reg reg const*		z	zcn
1111000	`or3`	*reg reg reg*			zcn
1111001	`or3i`	*reg reg const*		z	zcn
1111010	`xor3`	*reg reg reg*			zcn
1111011	`xor3i`	*reg reg const*		z	zcn
1111100	`asr3`	*reg reg shiftval*			zcn
1111101			reserved		
1111110			reserved		
1111111			reserved		